Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ЕВЧИК Мария Васильевна

ПРИКЛАДНЫЕ ВОЗМОЖНОСТИ ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ С ОГРАНИЧЕНИЯМИ
APPLIED CAPABILITIES OF CONSTRAINT LOGIC PROGRAMMING

Выпускная квалификационная бакалаврская работа
по направлению 030100 «Философия»

Научный руководитель –
доцент, к.филос.н.,
Ю. В. Нечитайлов

Студент:
Научный руководитель:
Работа представлена на кафедру
“___” _____ 2016 г.
Заведующий кафедрой:

Санкт-Петербург
2016

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Многие люди считают, что наука искусственного интеллекта направлена лишь на построение некой мыслящей машины, которая могла бы обладать схожими с человеческими чувствами и действовать подобным образом, так, чтобы сторонний наблюдатель мог сказать, что эта машина обладает разумом. На самом деле проблема гораздо шире и глубже. Большинство тем, касающихся искусственного интеллекта, актуальны так или иначе, ведь данная область является междисциплинарной и затрагивает не только «базовые» науки – философию, математику, механику, программирование.

На этот раз мы решили затронуть проблему на стыке двух областей: программирования и философской логики. Так как наука об искусственном интеллекте берет свое начало в философии, то нельзя даже предположить, что данной проблемой рано или поздно не заинтересуется столь строгая область, более того – самим предметом науки логики является мышление и законы, согласно которым оно осуществляется. Нельзя представить исследования разума без обращения к его инструменту, то есть мышлению. Однако одной логики не достаточно для построения мыслящей машины, ведь это чисто теоретическая наука, нуждающаяся в инструментарии, который она нашла в программировании.

Парадигма логического программирования базируется на идее, что мышление человека составляют суждения, и именно способность оперировать ими, выводить новые на основе уже имеющихся, причем так, чтобы они соответствовали законам мышления, - все это и делает человека разумным. Как логика, так и программирование имеют свои синтаксис и семантику (то есть знаки и их значения), определенные правила, - их интеграция стала базой, на которой и строились логические языки программирования. Более того, логическое программирование в своем

применении в дальнейшем вышло далеко за пределы искусственного интеллекта.

Собственно, именно этот выход и делает данную работу актуальной. Если предшествовавшие этой три работы были направлены на рассмотрение истории и путей развития науки об искусственном интеллекте, то предметом рассмотрения данной работы будет ее прикладная, практическая сторона. В ней наша цель – показать, как глубоко исследования и разработки, изначально служившие иным целям, вошли в нашу жизнь, полностью интегрировались со всеми ее сферами, стали незаменимыми. Если в отношении математики, механики в этом не было сомнения еще до создания самой науки искусственного интеллекта, то в отношении логического программирования это стало наиболее актуально за пятнадцать-двадцать лет.

К промежуточным задачам, связанным с такой целью, можно отнести:

- рассмотрение экспертных систем, их структуры и характеристик, которые, как мы считаем, являются наиболее прикладной ветвью области интеллектуальных систем, - этому будет посвящена первая глава

- разбор логических аспектов внутренней конструкции экспертных систем поможет нам выйти к логическим основам интересующей нас проблемы;

- рассмотрение во второй главе логического вывода и принципа резолюции, который во многом определил дальнейшее развитие логического программирования;

- изучение основных аспектов логического программирования, основ, истории его развития; основные направления, которые можно выделить, - все это составляет предмет третьей главы;

- описание некоторых проектов, представляющих собой реализацию логического программирования с ограничениями.

Мы считаем, что данной работе свойственна научная новизна и практическая значимость, так как литературы, которая охватывает обе области и рассматривает их взаимодействие именно под таким углом, очень мало. Таким образом, эту работу можно назвать определенным теоретическим вкладом в рамках данной области исследований.

ГЛАВА I. ИНТЕЛЛЕКТУАЛЬНЫЕ СИСТЕМЫ

Так как предмет нашего исследования – агенты, или системы, - основаны на знаниях, то их можно назвать интеллектуальными, то есть умеющими решать задачи. В истории развития таких интеллектуальных систем можно выделить три наиболее значимых этапа:

1) 60-70гг. – наука искусственного интеллекта только начинает развиваться, однако исследователи уже начинают понимать весь ее потенциал. Формируется социальный заказ на поддержку процессов принятия решений и управление. В ответ на это ученые создают персептрон (первые нейронные сети), разрабатывают метод эвристического программирования и ситуационного управления большими системами (СССР);

2) 70-80гг. – исследователи наконец обратили свои взоры на системы на основе знаний и поняли их важность для принятия адекватных решений; разрабатываются первые ЭС на основе нечеткой логики, первые модели правдоподобного вывода и рассуждений;

3) 80-90гг. – наука приходит к созданию гибридных моделей представления знаний, сочетающих в себе интеллекты: поисковый, вычислительный, логический и образный.

Преимущество использования интеллектуальных систем, или агентов, на основе знаний заключается в их способности заменить живых экспертов в разных прикладных областях. Дело в том, что знания способствуют формированию правильного поведения агентов. Тем не менее, их знания выражены в достаточно общих формах, которые каждый раз изменяются в зависимости от информации, поступающей из внешней среды. Это помогает системе выявлять скрытые аспекты текущего состояния каждый раз до того, как действовать.

Системы на основе знаний быстро накапливают опыт и достигают определенной компетентности, так как могут принимать новые задачи, получать новые знания и обновлять в соответствии с ними уже имеющиеся старые. У такого рода систем все знания представлены посредством логики, а у логических агентов знания всегда определенные: высказывание о мире либо истинно, либо ложно, хотя агент может и не знать о существовании некоторых высказываний.

1.1. Экспертные системы

Примером внедрения интеллектуальных систем в нашу повседневную жизнь будут экспертные системы – те, что созданы на базе ЭВМ и работа которых сводится к имитации умственной деятельности человека при решении сложных или творческих задач. Соответственно, нас интересуют такие ЭС, которые построены на знаниях, а не данных, хотя существуют и такие.

В семидесятых годах XX века в области ИИ сформировалась самостоятельное направление, занимавшееся разработками экспертных систем. Цель данных исследований заключалась в разработке таких программ или устройств, которые при решении задач не уступали бы в возможностях и способностях живому человеку-эксперту. По большей части такие системы решают не поддающиеся алгоритмизации задачи. В основе функционирования ЭС лежит использование знаний, а манипулирование ими осуществляется с помощью эвристических правил, сформулированных экспертами. Экспертные системы играют важную роль в нашей повседневной жизни: проводят классификацию, дают советы, ставят диагнозы и выполняют множество других автоматизированных задач. Отличие их от обычных машин на основе алгоритмического анализа состоит в том, что они специализируются на узкой области, используя метод дедуктивных рассуждений.

На сегодняшний день основными областями применения ЭС являются:

- медицина;

- электроника;

- вычислительная техника;

- геология;

- математика;

- космос;

- сельское хозяйство;

- управление;

- финансы;

- юриспруденция;

и т.д.

Почему мы обратились именно к экспертным системам? Ответ кроется в области их применения. Дело в том, что существуют такие задачи, которые решаются экспертными системами, основанными на знаниях, гораздо успешнее, чем любыми другими программами, и нас интересуют именно задачи первого типа.

Чтобы понять, что именно для данной проблемы необходим такой подход в решении, используются следующие критерии:

- данные и знания, которыми оперирует система, достоверны и неизменны;

- пространство возможных решений ограничено;

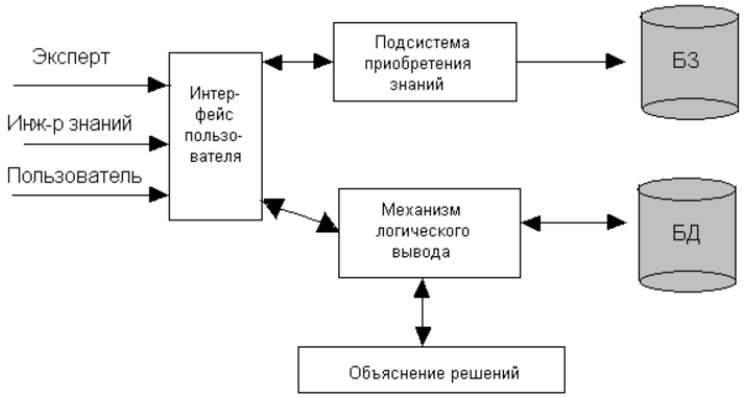
- задачи решаются с использованием формальных рассуждений. (Не стоит забывать, что существуют также задачи, которые целесообразнее решать с

помощью основанных на процедурном анализе систем. Нами рассматриваемые ЭС более пригодны для решения задач, где нужны формальные рассуждения.);

- должен быть как минимум один человек-эксперт, который может четко сформулировать свои знания и объяснить методы их применения.

1.1.1. Типовая структура экспертных систем

Ниже на рисунке мы видим упрощенную схему структуры ЭС; представленные блоки всегда находятся в неизменном порядке.



ЭС можно разделить на две категории, которые зависят от категории пользователей, соответственно имеются два различных «входа»:

1) пользователь – эксперт, которому нужна консультация системы. Диалог с ЭС происходит через диалоговый процессор. Существуют два типа диалога:

- на ограниченном подмножестве естественного языка с использованием словаря-меню;
- на основе нескольких возможных действий.

2) эксперты в определенной области и инженеры знаний составляют экспертную группу инженерии знаний, в функции которой входит заполнение базы знаний (Knowledge Base, KB). Данный процесс

осуществляется с помощью специализированной подсистемы приобретения знаний, что позволяет частично автоматизировать этот процесс.

Кратко рассмотрим каждый из этих блоков, и начнем с 1. Интерфейса пользователя. Он представлен системой программных и аппаратных средств, которые обеспечивают конечному пользователю использование компьютера для решения возникающих в его профессиональной среде задач, требующих экспертной консультации.

2. База знаний – один из самых важных компонентов ЭС. Предназначена для долгосрочного хранения данных, касающихся определенной, чаще всего узкой прикладной предметной области. Отличие этого компонента ЭС от других в том, что он поддается изменению, в процессе использования ЭС может пополняться новыми данными. Знания представлены в символьной форме, что необходимо для реализации символьной природы рассуждений (процесс рассуждения представлен как последовательность символьных преобразований), - об этом мы еще поговорим позже.

Существуют две разновидности базы знаний:

- динамическая: изменяется со временем. Содержимое зависит от состояния внешнего мира. Добавляемые в базу новые факты суть результат немонотонного вывода, состоящего в применении фактов к имеющимся знаниям. Данные в базе могут удаляться или изменяться в процессе решения задачи;

- статичная: характеризуется монотонным выводом, а факты не изменяются в процессе решения задач.

3. Промежуточная память ЭС представлена базой данных, в которых хранятся исходные и промежуточные данные решаемой задачи.

4. Сообщение между интерфейсом пользователя и базой знаний осуществляется подсистемой приобретения знаний. Как уже стало понятно из ее названия, она предназначена для добавления в КВ новых правил и модификации имеющихся. В первую очередь она приводит правило к необходимому виду, а затем применяет его в процессе работы. В сложных системах используются средства проверки добавленных и измененных правил на предмет противоречивости с уже имеющимися.

5. Пожалуй, самый важный компонент ЭС, ее основа, представлен механизмом логического вывода, который одновременно использует базу знаний и базу данных. Используя информацию из первой, она производит рекомендации для решения задачи. Чаще всего для представления знаний используются системы продукций и семантические сети. Для наглядного представления работы механизма логического вывода и того, как он влияет на функционирования всей ЭС в целом, рассмотрим такой пример:

База знаний состоит из фактов и правил (предположим, если <посылка>, то <заключение>). Если экспертная система определяет, что посылка верна, то правило признается подходящим, и оно запускается в механизм действия. Запуск правила означает тот факт, что заключение данного правила было включено в составную часть процесса консультирования. Цель ЭС – в результате применения правил прийти к некому уже установленному заключению (целевое заключение) и внести его в рабочее множество, либо опровергнуть его. В последнем случае доказывается его ложность, так как вывести его невозможно. Целевое заключение может быть либо изначально занесено экспертом в базу знаний, либо получено в ходе диалога с пользователем. Таким образом, работа экспертной системы представляет собой цепочку шагов, на каждом из которых из базы выбирается некоторое правило, которое затем применяется к рабочему множеству; работа заканчивается по достижении целевого утверждения (либо его опровержении). Такой цикл обработки правил и

называется логическим выводом. Есть два наиболее используемых способа проведения логического вывода:

- в прямом порядке;

- в обратном порядке.

Интересен последний способ. В случае проведения логического вывода в обратном порядке заключения рассматриваются до тех пор, пока не будут обнаружены в рабочей памяти или получены от пользователя факты, подтверждающие одно из них. Вначале выдвигается некоторая гипотеза, а затем механизм логического вывода, возвращаясь назад, переходит к фактам и ищет подтверждающий. Если гипотеза верна, то выбирается следующая, детализирующая первую, и так далее. Данный вывод называется управляемыми целями и используется в том случае, когда последние известны и их немного.

6. Еще один компонент ЭС, напрямую взаимодействующий с механизмом логического вывода – это объяснительный компонент. Суть его заключается в объяснении, как система получила решение и какие при этом использовались знания. Такая работа имеет сразу два положительных следствия: облегчает работу эксперту и повышает доверие пользователя к конечному результату.

Реализация экспертных систем предполагает использование входной информации в виде битов и байтов, однако на выходе информация должна быть представлена в виде естественного языка для осуществления диалога с пользователем, что требует дополнительных средств. Большинство систем имеет довольно примитивный интерфейс на естественном языке, когда допустимые входные сообщения ограничены набором понятий, заложенных в базу знаний.

1.1.2. Функционирование экспертной системы

Каждая ЭС может работать в двух режимах:

- в режиме приобретения знаний;

Отличительная черта экспертных систем в том, что программу разрабатывает не программист, а пользователь-эксперт, навыками программирования не обладающий. Он описывает проблемную область как совокупность правил и данных, последние же определяют объекты, их характеристики и значения в экспертной области. Правила определяют способы манипулирования данными в зависимости от выбранной области. Используя компонент приобретения знаний, эксперт наполняет ими систему. Новые знания позволяют системе самостоятельно решать задачи.

- в режиме консультации.

В данном случае с системой общается обычный пользователь, который может обладать или не обладать необходимыми навыками в интересующей его сфере. В режиме консультации данные о задаче через интерфейс пользователя поступают в рабочую память, в которой хранятся промежуточные данные о решаемой задаче. На основании входных данных из рабочей памяти, общих данных о проблемной области и правил базы знаний с помощью компонента логического вывода формируется решение задачи. В процессе решения ЭС сама формирует и исполняет определенную последовательность операций.

1.2. Знания

Напомним, что знания представляют собой определенные принципы, связи, законы, позволяющие решать задачи в заложенной в ЭС предметной области.

Представление знаний – это соглашение экспертов о том, как описывать реальный мир. Само описание происходит таким образом: в первую очередь на естественном языке вводятся основные понятия и

отношения меду ними, однако при том условии, что они уже давно известны. Затем подбирается подходящая символьная модель и происходит синхронизация меду ней и заданными характеристиками понятий. Цель представления знаний заключается в построении символьной модели реального мира и его частей. Средства, позволяющие описать реальный мир на языке представления, называют системой представления знаний (СПЗ). Они также помогают организовать хранение знаний в системе, их накопление, анализ, обобщение и структурирование, находить требуемые знания, выводить новые и объединять с уже имеющимися, из них выводить новые данные, налаживать интерфейс «пользователь – данные».

В системе знаний отдельные факты, характеризующие объекты, процессы и явления в предметной области, их свойства – все объединяются в одну группу и называются данными.

Можно выделить три основных свойства знаний. В первую очередь, они характеризуются внутренней интерпретируемостью. Каждая информационная единица должна иметь собственное имя, по которому программа могла бы найти ее. Если данные лишены имени, то система не может их идентифицировать. В таких случаях данные могут быть извлечены программой только по запросу самого программиста. Машина выдает лишь двоичный код, но что за ним скрыто, ей не известно.

Второе свойство – структурированность. Информационные единицы должны обладать гибкой структурой и подчиняться рекурсивному «принципу матрешки», то есть одна вложена в другую. Это дает возможность произвольно устанавливать отношения между отдельными информационными единицами («часть – целое», «род – вид», «элемент – класс»).

Следующая характеристика – связность. Связи между информационными единицами призваны характеризовать отношения между

данными единицами: «одновременно», «причина и следствие», «быть рядом». Они относятся к декларативному знанию. Процедурное знание, связанное с вычислением функций, характеризуется отношением «аргумент – функция» между двумя единицами. Существует множество разных отношений, выполняющих ту или иную роль. Отношения структуризации задают иерархии информационных единиц, функциональные отношения несут процедурную информацию, каузальные призваны задавать причинно-следственные связи, а семантические соответствуют всем остальным отношениям.

Необходимым условием функционирования ЭС также является активность системы знаний. С того момента, как появились ЭВМ и информационные единицы были разделены на данные и команды, возникла ситуация, при которой первые пассивны, а вторые активны. Протекающие в машине процессы инициируются командами, а данные используются только в случае необходимости. Проблема в том, что для интеллектуальных систем такая ситуация неприемлема. В ней знания способствуют актуализации тех или иных действий. Это нужно для того, чтобы выполнение программ в системе могло инициироваться текущим состоянием информационной базы. Появление в базе фактов, описаний событий, установление связей становится источником активности для системы.

Ниже представлена общая классификация знаний:

- поверхностные – знания о видимых взаимосвязях между отдельными событиями и фактами предметной области;

- глубинные – абстракции и схемы, которые отображают структуру и процессы, проходящие в предметной области.

1.2.1. Представление знаний

Представление знаний – одна из наиболее характерных для интеллектуальных систем проблем. Корень ее в том, что способ представления оказывает ощутимое влияние на характеристики и свойства всей системы. Для манипулирования знаниями о реальном мире в первую очередь нужно их смоделировать и перевести на язык представления. Используемые человеком знания важно отличать от тех, что использует компьютер.

При формировании модели представлений знаний всегда учитываются такие факторы, как однородность представления и простота понимания. Первое приводит к упрощению управления процессом логического вывода и знаниями, просто представление знаний также более понятно экспертам и простым пользователям, в противном случае затрудняется получение и обработка знаний. Тем не менее, это условие на практике не так легко выполнимо. Для решения простых задач выбирают среднее, компромиссное решение, тогда как решение больших и сложных задач требует структурирования и модульного представления.

Существует несколько моделей представления знаний, однако мы рассмотрим именно логическую.

В логическом представлении знаний традиционно фигурируют модели, основанные на классическом исчислении предикатов первого порядка, когда предметная область или задача описывается набором аксиом. Преимущество такого способа состоит в возможности непосредственно запрограммировать любой обладающий хорошими математическими свойствами мощный механизм. Такие программы вкупе с полученными ранее знаниями дают новые знания.

В чем преимущество логической модели?

- в ней всегда присутствуют регулярные методы вывода, в терминах которого определяются процедуры доказательств;

- можно использовать семантику, которая истолковывается разными способами в зависимости от целей логического представления. Например, декларативная семантика описывает предметную область, а процедурная семантика помогает понять, как выводятся новые высказывания из имеющегося набора формул;

- логическая модель представления характеризуется простотой, лаконичностью и единообразием употребляемой нотации.

Данная модель также известна своей наибольшей математической строгостью среди прочих. Тем не менее, на практике она не получила должного распространения из-за малой наглядности базы знаний. Основная область применения логической модели ограничивается учебными системами[4].

1.2.2. Процесс накопления знаний

Существуют определенные операции, с помощью которых добавляются новые знания в базу, и также извлекаются из нее. Они представлены двумя командами: *Tell* и *Ask*, которые связаны с процедурой логического вывода. На входе логический агент принимает результаты актов восприятия *percept*, а возвращает ответ-действие *action*. Команда *Tell* позволяет нам ввести новые данные *percept* в базу, тогда как с помощью команды *Ask* мы делаем запрос в базу о том, какое действие следует предпринять. Важно учесть, что логический вывод подчиняется тому фундаментальному требованию, что ответ системы должен соответствовать тем знаниям, которые ранее были сообщены базе.

Выполнение программы проходит в три этапа:

1) Tell: внесение данных актов восприятия percept в базу;

2) Ask: передача в базу запроса о том, какое действие предпринять;

- поиск ответа на запрос;

- процесс рассуждения о текущих состояниях мира, результатах возможных последствий действий;

3) Tell: регистрация агентом своего выбора;

- выполнение действия.

Датчики и исполнительные механизмы соединены тремя функциями:

1) Make-Percept-Sentence – формирует высказывание о том, что агент получил результаты конкретного акта в текущий момент времени;

2) Make-Action-Query – формирует высказывание-запрос о том, какое действие должно быть выполнено в данный момент;

3) Make-Action-Sentence – формирует высказывание о том, что выбранное действие было исполнено.

Каждая из трех функций выполняется на трех шагах программы соответственно.

Function KB-Agent (percept) **returns** действие *action*
static: KB, база знаний
t, счетчик, обозначающий время, первоначально равный 0

Tell(KB, Make-Percept-Sentence(percept, *t*))
action ← Ask(KB, Make-Action-Query(*t*))
Tell(KB, Make-Action-Sentence(*action*, *t*))
t ← *t* + 1
return *action*

До того момента, как агент начинает наполнять свою базу знаний полученными данными актов восприятия, его первоначальная программа создается путем добавления знаний проектировщика, так же в форме высказываний. Если к этому прилагается такой язык представления, который

способен выражать новые полученные знания в форме высказываний, то работа агента упрощается – это декларативный подход. Суть процедурного подхода сводится к представлению знаний непосредственно в виде кода программы.

Некоторые продвинутые агенты кроме стандартных операций представления знаний имеют также механизмы обучения. Они обеспечивают самостоятельное получение агентом общих знаний о среде с помощью актов восприятия, причем полученные данные затем заносятся в его базу знаний. Такие манипуляции позволяют агенту стать автономным.

1.2.3. Формирование простой базы знаний на примере мира робота-уборщика

Для наглядности рассмотрим модель мира логического. За основу примера возьмем мир вампуса из книги Рассела, Норвига [6]. Мы сохраним его структуру, но изменим составляющие.

Итак, у нас есть:

1) логический агент – робот-уборщик, задача которого – ликвидировать загрязнение, избежав цепких лап кота;

2) среда – комната, разделенная на квадраты, по которым перемещается агент, по 4 с каждой стороны;

3) цель – убрать загрязнение, которое может находиться в любой части комнаты, выдает себя неприятным запахом в близлежащих квадратах;

4) помеха – кот, агент не должен с ним столкнуться, кот выдает себя звуком “мяу” в близлежащих квадратах;

5) механизм действия – за один шаг агент преодолевает не более одного квадрата, может поворачиваться на 90% и не может двигаться по диагонали.

Для удобства использования сократим имеющиеся у нас части задачи и получим:

A – агент, робот-уборщик (agent)

C – кот (cat)

T – загрязнение (trash)

B – неприятный запах (bad smell)

M – “мяу” (“mew”)

OK – безопасный квадрат

1,4 smell	2,4	3,4	4,4
1,3 **trash**	2,3 smell	3,3	4,3
1,2 smell	2,2	3,2 “mew”	4,2
1,1 **agent**	2,1 “mew”	3,1 **cat**	4,1 “mew”

На схеме видим мир робота-уборщика, каждый квадрат его среды пронумерован. Агенту нужно сделать шаг и на основе полученных данных решить, близок ли он к своей цели либо к угрозе.

Как и любой агент, основанный на знаниях, робот-уборщик также имеет базу знаний. Чтобы робот смог сам сформировать логические выводы по всем правилам, нам в первую очередь нужно снабдить его знаниями из внешнего мира.

Для начала зададим словарь пропозициональных символов. Рассмотрим высказывание $C_{k,l}$ как истинное, если в квадрате [k, l] находится кот; $M_{k,l}$ истинно, если в квадрате [k, l] робот слышит присутствие кота.

В квадрате [1,1] нет кота, так как там находится сам агент. Тогда:

P_1: $\neg C_{1,1}$

Так как робот определяет близость врага по звуковому сопровождению, то в квадратах [2,1], [2, 4], [4, 1] должен быть звук “мяу”. Теперь мы можем построить первое сложное высказывание, литералы которого связаны конъюнкцией.

P_2: $C_{3,1} \leftrightarrow M[2,1] \cup M[2,3] \cup M[4,1]$

Наш робот находится в квадрате [1,1], и ближайший источник звука для него – квадрат [2,1]. Робот может его услышать, если переместится в этот квадрат:

P_3: $A_{2,1} \cap M_{2,1}$

Это простая схема конъюнктивных выражений, отражающих знания о мире. В базе знаний она будет иметь вид $(P_1 \cap P_2 \cap P_3)$, так как все высказывания истинные.

ГЛАВА II. ЛОГИЧЕСКИЕ АСПЕКТЫ ПРОЕКТИРОВАНИЯ ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ

2.1. Логический вывод

Обратимся к понятию логического следствия. В математике принято обозначение

$\alpha \models \beta$

которое означает, что высказывание α влечет за собой β, при этом истинность второго «содержится» в истинности первого. На основе процедуры логического следствия формируются заключения, то есть логический вывод. Агент использует такой ход для принятия решений и совершения действий, при этом высказывание α строится на основании уже имеющихся у него знаний, либо после обработки данных из внешней среды и сопоставления их с уже имеющимися знаниями (формирования базы знаний).

«Чтобы лучше понять, что такое логическое следствие и логический вывод, можно представить себе, что множество всех логических заключений, полученных из базы данных, — это стог сена, а высказывание α — это иголка. Логическое следствие напоминает утверждение о том, что в стоге сена есть иголка, а логический вывод можно сравнить с ее поиском. Это различие между понятиями логического следствия и логического вывода отражено и в формальных обозначениях: если некоторый алгоритм логического вывода i позволяет вывести логическим путем высказывание α из базы знаний KB, то можно записать следующее:

$$KB \vdash_i \alpha$$

Эта формула читается так: «высказывание α получено путем логического вывода из базы знаний KB с помощью алгоритма i» или

«алгоритм i позволяет вывести логическим путем высказывание α из базы знаний KB» [Рассел, Норвиг 2006, 292-293].

Вот несколько свойств, необходимых для успешного логического вывода:

1) непротиворечивость – выполняется при получении алгоритмом логического вывода только таких высказываний, которые действительно следуют из базы знаний;

2) полнота – алгоритм позволяет вывести любое высказывание, которое следует из базы знаний.

Как мы помним, язык представления описывает факты реального мира, поэтому важной проблемой для нас является соотнесения этих двух элементов. Коротко можно описать принцип их взаимодействия так: если база знаний истинна в реальном мире, то также истинно в реальном мире вытекающее из нее высказывание α, при условии, что оно получено логическим путем с помощь непротиворечивой процедуры логического вывода. Процесс логического вывода является частью реального мира, хоть нам и известно, что он работает только с синтаксисом. Однако невозможно построить семантику, если нет базы в виде синтаксиса. Тем не менее, мы все равно сталкиваемся с проблемой обоснования, а именно: как доказать, что база знаний истинна в реальном мире? Ответ кроется в связи агента с окружающей средой, которая создается с помощью датчиков.

2.2. Правило резолюции

Проведенный по правилам процесс логического вывода полностью совпадает с поиском решений в задачах поиска. Такой логический вывод называется доказательством. Он проводится по определенным шаблонам построения рассуждений: известный всем Modus Ponens, правило удаления конъюнкции, все правила логической эквивалентности и вытекающее из них

правило удаления двусторонней импликации, а также правило де Моргана. Такой способ используется как альтернатива перебору моделей.

Рассмотрим метод построения логического вывода, который лежит в основе обычного способа обработки логических программ. В данном методе используется только одно правило вывода, которое называется правилом резолюции, а каждое его применение – шагом вывода [Хоггер 1988, 32].

Начнем с примеров из мира робота-уборщика. Агент переместился в квадрат [2,1] и услышал находящегося рядом кота:

$P_4: M_{2,1} \leftrightarrow (C_{2,2} \cup C_{3,1})$

Так как в квадрате [1,1] робот звука не слышал, то кот в квадрате [1,2] быть не может:

$P_5: \neg C_{1,2}$

Предположим, что агент, дабы избежать столкновения с котом, перемещается обратно в квадрат [1,1]. Так как в нем нет звука “мяу” и плохого запаха

$P_6: \neg B_{1,1}$

то агент перемещается в квадрат [1,2]. Зато здесь он сталкивается с неприятным запахом, но не слышит звука “мяу”, что исключает наличие кота в квадрате [2,2] и делает его безопасным:

$P_6: \neg M_{1,2} \rightarrow \neg C_{2,2}$

$P_7: B_{1,2} \leftrightarrow (T_{1,3} \cup T_{2,2})$

Далее логика рассуждения такова: если в квадрате [1,2] нет звука “мяу”, но есть плохой запах, то в соседнем квадрате [2,2] нет кота, но может быть мусор. Соответственно, если робот перейдет в этот квадрат, то найдет там либо звук “мяу”, либо мусор, либо пустое поле. Также робот может перейти в

квадрат [1,3]. Однако робот выбирает неверный путь (о чем знаем только мы, но не агент) и идет в квадрат [2,2]. Не обнаружив там мусора и звука кота

P_8: $\neg T_{2,2}$

P_9: $\neg M_{2,2}$

он приходит к выводу, что мусор все же в квадрате [1,3], куда и направляется, снова пройдя через квадрат [1,2]:

P_{10}: $T_{1,3}$

В выражении P_4 мы предположили, что от может быть в квадратах [2,2] или [3,1], тогда:

P_{11}: $C_{2,2} \leftrightarrow (M_{2,1} \cup M_{1,2} \cup M_{2,3} \cup M_{3,2})$

что противоречит выражению P_6 и P_4. Таким образом, если в квадрате [2,2] нет кота, то мы выводим, что угроза находилась в квадрате [3,1]:

P_{12}: $C_{3,1}$

Все это большое рассуждение можно выразить кратко: если по крайней мере в двух квадратах [2,2] или [3,1] есть кот, но его нет в квадрате [2,2], то он есть в квадрате [3,1].

Приведение промежуточных высказываний к противоречию друг другу и получению с помощью правил вывода новых высказываний является примером использования правила резолюции и называется единичной резолюцией:

$$\frac{b_1 \cup \ldots \cup b_k, \;\; d}{b_1 \cup \ldots \cup b_{i-1} \cup b_{i+1} \cup \ldots \cup b_k}$$

где каждое выражение b – литерал, а d – отрицание литерала b_i (то есть они взаимно обратные литералы).

> *«Таким образом, в правиле единичной резолюции берется выражение (дизъюнкция литералов) и еще один литерал, после чего формируется новое выражение. Обратите внимание на то, что этот единственный литерал может рассматриваться как дизъюнкция из одного литерала, называемая также единичным выражением»* [Рассел, Норвиг 2006, 307].

Правило полной резолюции имеет вид:

$$\frac{b_1 \cup \ldots \cup b_k,\ \ d_1 \cup \ldots \cup d_n}{b_1 \cup \ldots \cup b_{i-1} \cup b_{i+1} \cup \ldots \cup b_k \cup d_1 \cup d_{j-1} \cup d_{j+1} \cup \ldots \cup d_n}$$

здесь литералы b_i и d_j являются взаимно обратными. Упрощенная запись этого правила выглядит так:

$$\frac{b_1 \cup b_2,\ \ \neg b_2 \cup b_3}{b_1 \cup b_3}$$

Смысл правила резолюции в том, что берутся два выражения и получается новое, производное от них, которое содержит все литералы исходных выражений, кроме двух взаимно исключающих литералов.

Важным условием построения логического вывода с помощью правила резолюции является применение операции факторизации, которая состоит в удалении дополнительных копий литералов. Это необходимо для того, чтобы конечное высказывание содержало только одну копию каждого литерала.

Как мы помним, одна из главных характеристик логического вывода – его непротиворечивость, соответственно, правило резолюции также должно быть непротиворечивым. Это можно доказать, снова рассмотрев литерал b_i. Если он истинен, то литерал d_j ложен, тогда выражение $d_1 \cup d_{j-1} \cup d_{j+1} \cup \ldots \cup d_n$ тоже должно быть истинным, так как известно, что истинно $d_1 \cup \ldots \cup d_n$. Мы также знаем, что истинен литерал b_i, тогда истинно все выражение $b_1 \cup \ldots \cup b_{i-1} \cup b_{i+1} \cup \ldots \cup b_k$, ведь выражение $b_1 \cup \ldots \cup b_k$

истинно. Таким образом, b_i является либо истинным, либо ложным, а один из этих выводов утверждается в правиле резолюции.

Вспомним еще одну характеристику логического вывода – полноту. Правило резолюции создает основу для семейства полных процедур логического вывода.

> *«Любой полный алгоритм поиска, в котором применяется только правило резолюции, позволяет вывести любое заключение, которое следует из любой базы знаний в пропозициональной логике»* [Рассел, Норвиг 2006, 308].

Однако полноту этого правила стоит понимать только в узком смысле. Если высказывание A истинно, то правило резолюции нельзя использовать для получения A ∪ B, зато с его помощью можно проверить истинность этого выражения. Такое свойство резолюции называется полнотой опровержения, суть его заключается в том, что данное правило может быть использовано только для подтверждения либо опровержения высказывания, третьего не дано (то есть с его помощью невозможен перебор всех истинных высказываний).

2.3. Конъюнктивная нормальная форма

Выше мы рассмотрели, как правило резолюции применяется для дизъюнкций литералов, однако может ли оно служить правилом вывода для всей пропозициональной логики, а не только баз знаний? Ответ заключается в том, что «высказывание пропозициональной логики логически эквивалентно конъюнкции дизъюнкций литералов. Любое высказывание, представленное как конъюнкция дизъюнкций литералов, называется высказыванием, находящимся в конъюнктивной нормальной форме, или CNF (Conjunctive Normal Form)» [6].

Семейство таких высказываний ограничено и называется высказываниями в форме k-CNF, так как такие высказывания имеют точно k литералов в расчете на каждое выражение:

$(b_{1,1} \cup \ldots \cup b_{1,k}) \cap \ldots \cap (b_{n,1} \cup \ldots \cup b_{n,k})$

Предлагаем проиллюстрировать эту процедуру, преобразовав высказывание $P_4: M_{2,1} \leftrightarrow (C_{2,2} \cup C_{3,1})$ в конъюнктивную нормальную форму.

1) в первую очередь устраняем связку ↔, заменив высказывание $\alpha \leftrightarrow \beta$ на $(\alpha \rightarrow \beta) \cap (\alpha \rightarrow \beta)$:

$$(M_{2,1} \rightarrow (C_{2,2} \cup C_{3,1})) \cap ((M_{2,1} \rightarrow (C_{2,2} \cup C_{3,1}))$$

2) вторым шагом устраняем связку →, заменив $\alpha \rightarrow \beta$ на $\neg\alpha \cup \beta$:

$$(\neg M_{2,1} \cup C_{2,2} \cup C_{3,1}) \cap (\neg(C_{2,1} \cup C_{3,1}) \cup M_{2,1})$$

3) CNF требует, чтобы связка ¬ была только перед литералами, поэтому “введем связку ¬ внутрь выражения” с помощью эквивалентностей:

$\neg(\neg \alpha) \equiv \alpha$ (устранение двойного отрицания)

$\neg(\alpha \cap \beta) \equiv (\neg\alpha \cup \neg\beta)$ (правило де Моргана)

$\neg(\alpha \cup \beta) \equiv (\neg\alpha \cap \neg\beta)$ (правило де Моргана)

В нашем примере необходимо применить только последнее правило:

$(\neg M_{2,1} \cup C_{2,2} \cup C_{3,1}) \cap (\neg C_{2,2} \cup M_{2,1}) \cap (\neg C_{3,1} \cup M_{2,1})$

В итоге получилось выражение, приведенное к нормальной конъюнктивной форме как конъюнкция трех выражений. Его труднее прочесть, зато теперь к нему можно применить правило резолюции.

Далее рассмотрим подробнее алгоритм правила резолюции. Как уже было сказано, суть данного процесса в установлении противоречия, которое приводит к доказательству. Например, для доказательства выражения $(KB \equiv \alpha)$ необходимо доказать несостоятельность его взаимно обратного выражения $(KB \cap \neg\alpha)$.

В первую очередь приводим $(KB \cap \neg\alpha)$ к CNF. Затем резольвируем его (применяем правило резолюции) и удаляем все взаимно обратные выражения для получения нового, которое добавляется к уже существующим. Процесс резольвации происходит до тех пор, пока не произойдет одно из двух событий:

1) новые выражения перестают вырабатываться, тогда $\neg(KB \rightarrow \alpha)$;

2) в результате удаления взаимно обратных высказываний получается пустое выражение, тогда $KB \rightarrow \alpha$.

Пустое выражение равносильно высказыванию False, так как дизъюнкция истинна только при условии истинности одного из ее дизъюнктов.

Function PL-Resolution(*KB*, α) **returns** значение *true* или *false*
inputs: *KB,* база знаний – высказывание в пропозициональной логике
α, запрос – высказывание в пропозициональной логике
clauses ← множество выражений, полученное после преобразования высказывания $KB \cap \neg\alpha$ в форму CNF

new ← { }
loop do
for each C_i, C_j **in** *clauses* **do**
resolvents ← PL-Resolve (C_i, C_j)
if множество *resolvents* содержит пустое выражение
then return *true*
new ← *new* ∪ *resolvents*
if *new* ⊆ *clauses* **then return** *false*

clauses ← *clauses* ∪ *new*

Теперь вернемся к полноте правила резолюции и покажем, почему алгоритм PL-Resolution является полным: «Для этого целесообразно ввести понятие резолюционного замыкания RC(S) множества выражений S, представляющего собой множество всех выражений, которые могут быть получены путем повторного применения правила резолюции к выражениям из множества S или к их производным. Резолюционным замыканием является множество выражений, которое вычисляется алгоритмом PL-Resolution в качестве окончательного значения переменной clauses. Можно легко показать, что множество RC(S) должно быть конечным, поскольку количество различных выражений, которые могут быть сформированы из символов $P_1, \dots, P_k$, присутствующих в S, является конечным. (Следует отметить, что это утверждение не было бы истинным, если бы не применялся этап факторизации, в котором уничтожаются дополнительные копии литералов.) Поэтому алгоритм PL-Resolution всегда оканчивает свою работу» [Рассел, Норвиг 2006, 311].

В пропозициональной логике теорема полноты для правила резолюции называется основной теоремой резолюции.

> *«Если множество выражений является невыполнимым, то резолюционное замыкание этих выражений содержит пустое выражение»* [Рассел, Норвиг 2006, 311].

Эта теорема доказывается путем применения к ней правила резолюции и доказательства взаимно обратной ей: если замыкание RC(S) не содержит пустое выражение, то множество S выполнимо. Создание истинностной модели $P_1, \dots, P_k$ для множества S выглядит таким образом:

Для *i* от 1 до *k*:

- если в множестве S есть выражение, имеющее в себе литерал $\neg P_i$, и все другие литералы при выбранном для $P_1, \dots, P_{i-1}$ присваивании оказываются ложными, то литерал P_i ложен;

- в противном случае литерал P_i истинен;

- если множество RC(S) согласно правилу резолюции не является замкнутым и не содержит пустого выражения, то вышеуказанные присваивания представляют собой модель множества выражений S.

Теперь обратимся к тем самым хорновским выражениям. Это выражения в ограниченной форме, представляющие собой дизъюнкцию литералов, среди которых положительным является только один. Это происходи потому, что во многих практических ситуациях зачастую не используется вся мощь правила резолюции.

Представим, что нам дана некая база знаний. Мы можем использовать алгоритм прямого логического вывода PL-FC-Entails? (KB, q) для определения того, является ли пропозициональный символ q следствием из этой базы. Алгоритм начинает свою работу с уже известных фактов в базе знаний. Следствие некоторой импликации добавляется к базе знаний в том случае, если известны все ее антецеденты. Такой процесс остановится в двух случаях:

- если в базе знаний появился пропозициональный символ q;

- дальнейший логический вывод невозможен.

Алгоритм PL-FC-Entails? Действует за время, определенное линейной последовательностью.

Function PL-FC-Entails? (KB, q) **returns** значение *true* или *false*

inputs: *KB* – база знаний, множество пропозициональных хорновских выражений

q, запрос – пропозициональный смысл

local variables: *count*, таблица, индексированная по высказываниям, первоначально имеющая размер, соответствующий количеству предпосылок
inferred, таблица, индексированная по символам, в которой каждая запись первоначально имеет значение false
agenda, список символов, первоначально включает символы, известные как истинные в базе знаний *KB*

while список *agenda* не пуст **do**
p ← Pop(*agenda*)
unless *inferred[p]* **do**
inferred[p] ← *true*
for each хорновское выражение *c* в котором присутствует предпосылка *p* **do**
уменьшить на единицу значение *count[c]*
if *count[c]* = 0 **then do**
if Head[c] = q **then return** *true*
Push(Head[c], *agenda*)
return *false*

1

Непротиворечивость алгоритма PL-FC-Entails? Легко доказать, так как каждый шаг логического вывода является применением правила отделения. То же можно сказать и о полноте: в данном алгоритме может быть получено каждое атомарное высказывание, которое следует из KB. Рассмотрим конец таблицы inferred: она содержит значение true для каждого логически выведенного символа, и false – для остальных. Таблица inferred может быть рассмотрена как логическая модель, «более того, каждое определенное

---

[1] В данном алгоритме:
- список agenda – отслеживает истинные, но еще не “обработанные” символы;
- таблица count – отслеживает количество еще не известных антецедентов импликации.
При каждой обработке нового символа a из списка agenda в таблице count сокращается на одну единицу количество антецедентов для каждой импликации, имеющей в своих посылках символ a. Когда в таблице список антецедентов достигает 0, это значит, что все посылки импликации известны, поэтому ее заключение может быть добавлено к списку agenda. Логическим путем полученные символы нет смысла добавлять к списку «ожидания», если он уже был обработан. Это экономит время и служит профилактикой появления бесконечных циклов.

выражение в первоначальной базе знаний KB является истинным в данной модели» [Рассел, Норвиг 2006, 314].

Примером формирования управляемых знаниями логических рассуждений можно назвать прямой логический вывод. Это такие рассуждения, обработка которых начинается с уже известных данных. Он может использоваться в любом агенте на основе знаний для формирования заключений, основываясь на информации из внешнего мира, и зачастую для поступления некоторых данных даже нет какого-либо запроса. Для людей такой способ формирования чересчур «избыточен», так как количество поступающей извне информации превышает необходимое.

Что касается обратного логического вывода, то он действует в обратном направлении от полученного запроса. Если известно, что высказывание в запросе b истинно, то дальнейшая работа не проводится. Если это высказывание ложно, то алгоритм ищет все импликации, из которых вытекает b. Если консеквенты их истинны, то и b истинно.

Такой способ логического вывода является одной из форм рассуждения, направляемого целями, которая полезна при поиске ответов на конкретные вопросы. Чаще всего происходит так, что время, затраченное на обратный логический вывод, намного меньше количества времени, которое линейно зависит от размера базы [6]. В обратном логическом выводе затрагиваются только факты, непосредственно относящиеся к делу. Выработка агентом фактов помогает ему ограничивать работу прямого формирования рассуждений и таким образом отделять его работу от обратного логического вывода. Такие факты, по всей видимости, относятся к запросам, которые надлежит решать с помощью обратного логического вывода.

ГЛАВА III. ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

3.1. Язык логического программирования Пролог

Интенсивное (для относительно новой области) развитие компьютерных технологий, в особенности науки об искусственном интеллекте, в 1981 году привели к созданию Японским министерством международной торговли и промышленности новой исследовательской организации – Института по разработке методов создания компьютеров нового поколения (Institute for New Generation Computer Technology Research Center). Его основной задачей являлось создание основанных на знании систем обработки информации. Ученые исходили из идеи, что ими легче будет управлять, так как эти системы будут понимать естественный язык и, соответственно, за счет этого иметь возможность общаться с конечными пользователями. Основными характеристиками таких систем должны были быть:

- накопление знаний и использование их при решении задач (откуда вытекает следующее свойство);

- самообучаемость;

- предоставление пользователям экспертных консультаций;

- простота в управлении.

Вскоре после японского проекта также открылись подобные ему американский и европейский.

Так началась эра компьютеров пятого поколения, которые, в отличие от представителей предыдущего поколения, основывались не на сверхбольших интегральных схемах, но на распределенных вычислениях. Предполагалось, что появление таких систем изменит технологии за счет использования баз знаний. Цель, преследуемая японскими исследователями,

состояла не в подстраивании человеческого мышления к принципам работы ЭВМ, но наоборот – сама машина должна была приблизиться к человеческому мышлению. Предполагалось, что первая ЭВМ пятого поколения будет создана в 1991 году [2, 9].

Конечно, сейчас мы знаем, что многие из намеченных целей не были достигнуты, однако этот проект послужил толчком к новым разработкам в области интеллектуальных систем и подогрел интерес к логическому программированию. Разработчики отвергали традиционную фон Неймановскую архитектуру компьютера, что привело к созданию специализированных компьютеров для логического программирования PSI и PIM.

В качестве основной парадигмы для разработки ПО новых компьютеров был выбран язык логического программирования PROLOG. В английском языке его название расшифровывается как PROgramming in LOGic. Данный язык основывается на исчислении предикатов, а точнее на процедуре доказательства теорем методом резолюции для хорновских выражений.

Все началось со статьи Джона Робинсона «Машинно-ориентированная логика, основанная на принципе резолюции» (по-английски «A machine oriented logic based on the resolution principle»), которая в 1965 году была опубликована в 12 номере журнала «Journal of the ACM» [Робинсон 1970]. В ней Робинсон развил идеи своего предшественника Жака Эрбрана, который еще в 1931 году предложил так называемый «принцип резолюции», однако компьютеров тогда еще не было[5]. Робинсон расширил данный принцип, ввел идею унификации и сам алгоритм унификации. С помощью этого метода можно доказывать теоремы, сформулированные как формулы логики первого порядка, и, таким образом, в качестве ответов на вопросы получать «да» и «нет». Единственное, чего не хватало в предложенном Робинсоном принципе, - возможности вычислять ответы на вопросы [Apt 2007, ix].

Соответствующий шаг для преодоления этого ограничения был предложен Робертом Ковальски в его статье «Логика предикатов как язык программирования» («Predicate Logic as Programming Language», IFIP Congress), вышедшей в 1974 году. В этой статье Ковальский утверждает, что ограничение хорновскими выражениями даст более эффективный результат[5]. Основанный на таком решении проблемы подход стал известен как логическое программирование. Ряд других предложений, также имеющих своей целью вычисления с помощью логики предикатов первого порядка, появились примерно в то же время, но логическое программирование оказалось самым простым и универсальным вариантом [Apt 2007, ix].

Параллельно с этим «группа искусственного интеллекта» из Марсельского университета, состоящая из Алана Колмероэ и его коллег, работала над созданием нового языка программирования, который был бы основан на автоматическом доказательстве теорем, а в его задачи входила бы обработка естественного языка. В конечном итоге это все-таки привело к созданию первого прототипа вышеупомянутого языка Пролог в 1973 году. Программа, получившая название от PROgrammation en LOGique (фр.), была написана на Фортране и работала очень медленно[5]. Ковальски и Колмероэ с коллегами часто сотрудничали в период 1971-1973гг., и это объясняет тот факт, что их идеи во многом пересекаются: именно Пролог стал первой практической реализацией идеи логического программирования. Несмотря на то, что изначально он был спроектирован с целью обработки естественного языка, благодаря вкладу нескольких исследователей Пролог был преобразован в язык программирования общего назначения после своего выхода.

Однако Колмероэ, экспериментируя с Прологом, обнаружил некоторые важные его ограничения. Например, в нем можно было решать уравнения между термами (путем объединения), но не линейные уравнения, т.е. Пролог

поддерживал только одно решающее устройство. Это подтолкнуло Колмероэ к созданию серии преемников: Пролог I, Пролог III, Пролог IV. В частности, только Пролог III был расширен за счет возможности решения ограничений логических значений, вещественных чисел и списков, и может быть рассмотрен как первая реализация идеи программирования в ограничениях [Apt 2007, x].

Позже, в 1977 году, в Эдинбурге был создан первый компилятор языка Пролог, «отголоски» которого дошли до нас в последующих реализациях этого языка. Что интересно, он сам был запрограммирован на Прологе. Известная как «эдинбургская версия», эта реализация языка стала первым и единственным стандартом. На сегодняшний день редко можно встретить Пролог-систему, которая не поддерживает этот стандарт. Однако если такое случается, то в систему обязательно будет входить подсистема-переводчик программы в «эдинбургский» вид.

Первая версия Пролога для персональных ЭВМ была разработана Кларком и Майккебом в 1980 году, а в 1981 стартовал проект японского Института по разработке методов создания компьютеров нового поколения, о котором мы писали выше.

На сегодняшний день существует много реализаций языка PROLOG, созданных как российскими учеными, так и зарубежными. Вот наиболее известные из них:

Пролог-Д (Сергей Григорьев), Акторный Пролог (Алексей Морозов), а также Флэнг (А. Манцивода, Вячеслав Петухин) – в России; в других странах: BinProlog, AMZI-Prolog, *Arity* Prolog, Cprolog, *Micro* Prolog, Мпролог, Prolog-2, Quintus Prolog, SICTUS Prolog, Silogic *Knowledge Workbench,* Strawberry Prolog, SWI Prolog, UNSW Prolog и т. Д.[5].

До Пролога существовало еще несколько языков представления знаний: LISP (LISt Processing language) – самый первый и базовый язык, а точнее их семейство, созданное в начале шестидесятых и имевший большую популярность при разработке интеллектуальных систем и, в частности, экспертных систем. Примерно в то же время также разрабатывались другие языки программирования, основы которых существенно отличались от уже известных. К ним относились РЕФАЛ (Рекурсивных Функций Алгоритмический)[7], созданный в ИМП АН СССР, и SNOBOL (StriNg Oriented symBOlic Language), вышедший из лабораторий AT&T Bell Labs[9].

Выпущенный через десяток лет Пролог считался достойной и альтернативой своим предшественникам. В отличие от ЛИСПа, он не давал сверхмощных инструментов программирования, но позволял программисту не заботиться о потоке управления в программе. Особенность Пролога заключалась в той особой парадигме мышления, которую он предлагал: в ней задача описывалась в виде слабоструктурированной системы отношений, при этом количество последних не должно было превышать определенного количества (в противном случае это угрожало удобству использования), а каждое отношение должно было описываться небольшим числом альтернатив. Основным неудобством Пролога являлась его энергоемкость: механизм логического вывода использовал перебор всех возможных решений[1].

3.2. Программирование с ограничениями

Рассмотрим еще не затронутое нами программирование в ограничениях. Изначально понятие ограничений использовалось только в физике и комбинаторных вычислениях. Внедрением в свою терминологию нового понятия компьютерные науки были обязаны Айвену Сазерленду. Американский информатик считается пионером компьютерной графики, ведь именно он создал Скетчпад – первый прототип будущих систем

автоматизированного проектирования, который к тому же обладал также прототипом графического интерфейса. Сазерленд впервые использовал понятие «ограничения» в своей статье «Sketchpad: A man-machine graphical communication system» (1963). В ней он пишет, что по мере разработки компьютеризированной системы рисования были открыты и реализованы новые возможности: включение произвольных символов в чертеж, ограниченную возможность связывать части рисунка любым вычислимым способом, а также определение возможности копирования для построения сложных связей внутри комбинаций из простых атомарных ограничений. В свою очередь, под атомарными ограничениями Сазерленд понимал основные отношения между частями изображения, которые встроены в систему. Именно они отвечают за вертикальность, горизонтальность, параллельность, перпендикулярность линий на экране монитора, и так далее[19, 17]. Однако мы понимаем, что использование этого термина довольно специфично.

Используемое нами в рамках данной парадигмы понятие ограничений связано с упомянутыми выше ограниченными хорновскими выражениями. Более подробно они будут рассмотрены позже, однако для ясности нужно сказать, что это такие выражения, в которых присутствует не более одного положительного высказывания.

Считается, что в семидесятых годах несколько экспериментальных языков использовали идеи ограничений и опирались на концепцию решения ограничений. В это же время в области искусственного интеллекта была сформулирована проблема удовлетворения ограничений (CSP), что было вызвано разработками машинного зрения. Позднее еще одна концепция – концепция распространения ограничений – была признана наилучшим способом преодоления комбинаторного расширения, которое возникало при решении задач удовлетворения ограничений с использованием метода поиска сверху вниз.

Поиск методом сверху вниз – общее название для комплекса процедур поиска, в котором построение решения выглядит как систематические попытки расширить уже полученное частичное решение путем добавления ограничений. В простейших задачах каждое такое ограничение присваивает значение другой переменной. Большинство процедур такого метода поиска объединяет общий для них механизм возврата. Есть также еще один, - метод ветвей и границ, - то есть оптимизированный поиск сверху вниз, который впервые использовался в рамках комбинаторной оптимизации.

В восьмидесятых годах были предложены и реализованы первые языки программирования с ограничениями. Наиболее значимыми были те, что основывались на логической парадигме, так как включали в себя расширения логического программирования идеей ограничений. Основная причина успеха данного подхода заключалась в том, что программирование в ограничениях и предикаты логического программирования математически родственны. Первая реализация этой парадигмы уже была упомянута выше: это Пролог III, созданный Колмероэ. Само понятие программирования с ограничениями было введено в 1987 году. В это же время увидели свет операционная модель и семантика данного подхода, а также сформирована база для языка CLP®.

Еще одним ранним языком программирования в ограничениях был CHIP, разработанный в ECRC (European Computer-Industry Research Center, Мюнхен). Он включал в себя понятие удовлетворения ограничений внутри парадигмы логического программирования путем использования переменных, пробегающих ограниченные пользователем домены. Важное условие – значения ограниченных переменных не известны, только их текущие домены. Если переменная домена уменьшается до одного значения, то тогда оно – конечное значение переменной. CHIP тоже основывался на техниках поиска сверху вниз, которые пришли в программирование из области искусственного интеллекта.

3.3. Язык логического программирования с ограничениями ECLiPSe

ECRC был основан в 1984 году тремя европейскими компаниями для исследования развития передовых техник рассуждения, которые могли быть применимы к практическим проблемам. Были спроектированы и реализованы три системы программирования. Одна из них включала комплекс проблем, которые должны были решаться на многопроцессорных аппаратных средствах и в итоге на сети машин. Вторая поддерживала продвинутые техники интеллектуальной обработки баз данных в ресурсоемких приложениях. Третьей системой был CHIP. Все три системы были построены на основе логического программирования.

В 1991 году эти три системы были объединены, и появился ECLiPSe. Особенности данного языка, связанные с программированием в ограничениях, были изначально основаны на системе CHIP, которая к тому моменту ECRC выделила в отдельную компанию. В течение следующих 15 лет в ответ на просьбы пользователей были окончательно расширены инструменты решения ограничений, которые поддерживал Эклипс.

В 1997 году был выпущен первый интерфейс для внешнего ультрасовременного линейного и смешанного целочисленного пакета программирования. В 2000 году произошла интеграция инструментов решения в конечном домене и в линейном программировании, которые поддерживали смешанные алгоритмы. В 2001 была выпущена библиотека ic, которая поддерживала ограничения на основе логического типа данных, целые и вещественные числа, и удовлетворяла требованиям практического применения. Эклипс также включал в себя библиотеки некоторых языков программирования, например, CLP®, который был создан отдельно. В отличие от инструментов решения проблем, инструменты параллельного

программирования и баз данных в Эклипсе использовались намного меньше, и спустя годы многие подобные функции в системе были опущены.

Команда Эклипс была включена в список европейских исследовательских проектов, в особенности проект Esprit CHIC – Constraint Handling in Industry and Commerce (Обработка Ограничений в Индустрии и Коммерции) (1991-1994). После окончания исследовательской деятельности ECRC в 1996 году Эклипс был активно развит в Центре Планирования и Контроля Ресурсов (Centre for Planning and Resource Control) при Имперском Колледже Лондона (IC-Parc), при поддержке ICL (International Computers Ltd), Британского Совета Инженерных и Физических Исследований (the UK Engineering and Physical Sciences Research Council); должное развитие также получил проект Esprit CHIC-2 – Creating Hybrid Algorithms for Industry and Commerce (Создание Гибридных Алгоритмов для Индустрии и Коммерции) (1996-1999). Проекты Esprit сыграли важную роль в развитии языка Эклипс, обратив на него внимание исследователей. В частности, их вклад заключался в том, что они подчеркнули важность конечного пользователя, а также времени и навыков, необходимых для изучения программирования с ограничениями и разработки крупномасштабных эффективных и правильных программ.

В 1999 коммерческие права на Эклипс были переданы дочерней от IC-Parc компании Parc Technologies, которая использовала его в своих продуктах оптимизации и обеспечила финансирование для его защиты и дальнейшего развития. В августе 2004 Parc Technologies вместе с платформой Эклипс были приобретены компанией Cisco Systems.

На сегодняшний день Эклипс используется в сотнях институтов по всему миру для обучения и проведения исследований и все еще находится в свободном доступе для использования в образовании и исследовательских целях. Ему было найдено множество применений учеными со всего мира, в

том числе для планирования производства, транспортного планирования, биоинформатики, оптимизации контрактов и множества других целей.

Эклипс можно использовать для обучения большинству аспектов решения комбинаторных задач, таких как проблемы моделирования, программирование в ограничениях, математическое программирование и методы поиска. Он содержит:

- библиотеки решения ограничений;

- моделирование высокого уровня;

- язык управления (надмножество Пролог);

- интерфейсы для посторонних решающих устройств;

- интегрированную среду разработки.

С самого момента покупки и до настоящего времени Эклипс используется компанией Cisco для развития программного обеспечения коммерческой оптимизации [Apt 2007, xiii].

3.3.1. Особенности языка

Так как Эклипс разрабатывался на базе Пролога, для него характерна высокая степень совместимости с последним, даже с ISO Prolog – его стандартизированным диалектом. Эклипс относится к парадигме декларативного программирования (фокусируется на описании сущности проблемы и ее решения), что позволяет использовать его и как язык моделирования для описания проблем, и как язык общего назначения.

Помимо основных типов данных, которые были заимствованы из Пролога, также доступны и другие функции:

- строки (значениями строкового типа данных является произвольная строка - последовательность символов алфавита. Каждая переменная данного типа

может быть представлена фиксированным количеством байтов или иметь определенную длину[11]);

- неограниченное количество целых и рациональных чисел;

- числа с плавающей точкой (форма представления вещественных чисел).

Эклипс также поддерживает синтаксис массивов и структур с именами полей, что особенно необходимо в моделировании с ограничениями. Логическая конструкция итераций устраняет необходимость использования большинства простых рекурсивных паттернов (шаблонов проектирования в программировании).

Эклипс представляет обширный набор приспособлений для осуществления управляемого данными поведения. Они включают в себя декларативные пункты задержки и примитивные процедуры для мета-программного контроля (явная приостановка цели, гибкие средства вызова, приоритеты выполнения). Вместе с атрибутивным типом переменных данных такие приспособления становятся залогом функциональности многих расширений базового языка логического программирования, в том числе и тех, что основаны на ограничениях. Система вызывает определенные пользователем инструменты обработки событий, когда сталкивается с атрибутивными переменными в конкретных условиях, например, при унификации. Модульная система контролирует видимость предикатов, нелогические блоки памяти, исходные преобразования и настройки синтаксиса. Интерфейсы модуля могут быть расширены и ограничены, а модули, написанные на диалекте другого языка, могут быть смешаны в рамках одного приложения.

Программы могут содержать структурированные комментарии, из которых составляется справочная документация.

3.3.2. Библиотеки

Эклипс предоставляет несколько библиотек средств решения ограничений, которые могут быть использованы в нескольких приложениях:

- арифметические ограничения конечных областей;

- конечное множество ограничений;

- обобщенное распространение;

- интерфейсы внешних симплексных решающих устройств;

- правила обработки ограничений

и т.д.

Прочие библиотеки осуществляют поиск различными методами, например:

- поиск методом ветвей и границ;

- поиск, основанный на ремонте;

- поиск на основе ограниченного расхождения.

Эклипс согласовывается с внешними решающими устройствами, в особенности с линейными и смешанными целочисленными программными решающими устройствами (COIN-OR, CPLEX® и Xpress-MP).

Библиотеки совместимости для ISO Prolog и других диалектов Пролога (C-Prolog, Quintus, SICStus, SWI-Prolog) позволяют повторно использовать библиотеки, написанные на этих языках. Прочие вспомогательные библиотеки, включающие некоторое количество своих популярных публичных доменов, сами включены в дистрибутив.

3.3.3. Архитектура системы

Система включает в себя инкрементный компилятор, который переводит исходный код в виртуальный машинный код. Компилятор

оптимизирует выбор индекса, порядок унификации и встраивание контролирующих конструкций.

Исполнительная система реализует виртуальную машину, автоматическое управление памятью со сбором мусора из стеков и словаря, обработку событий и управляемое данными исполнение. Компоненты данного языка могут быть интегрированы в программное обеспечение через низкоуровневый интерфейс C или C++, а также высокоуровневые интерфейсы Java и Tcl.

ГЛАВА IV. ПРИМЕНЕНИЕ ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ С ОГРАНИЧЕНИЯМИ

Как уже было сказано, язык Эклипс широко используется в коммерческих и прикладных целях, в том числе и для проектирования рассмотренных выше экспертных систем. В данной главе мы рассмотрим несколько проектов, представляющих собой примеры реализации данного языка.

На сайте ECLiPSe[15] в открытом доступе представлено большое количество когда-либо реализованных проектов, к каждому из которых прикреплены описание, ссылка на сайт проекта, код и ссылка загрузки. Все проекты делятся на семь подгрупп: интернет вещей, язык, моделирование, OSGi (Open Services Gateway Initiative), время выполнения, тестирование, инструменты.

4.1. Интернет вещей

Для демонстрации этой категории мы выбрали не только действующий, но и успешно реализованный проект Concierge 5.0. Это компактная платформа OSGi (спецификация модульной системы и сервисной платформы для Java-приложений), нацеленная на реализацию проектов для встраиваемых устройств и IoT-приложения.

Concierge 5.0 направлен на полное соответствие с R5 OSGi для платформ, а также дополнительными элементами, такими как модуль запуска и сервис дополнительного обслуживания. Платформа может использоваться как самостоятельный распознавательный пакет (совместимый с сервисным внедрением решающего устройства).

Помимо самой платформы, данная версия включает в себя несколько дополнительных опциональных пакетов, например, минимальная реализация

оболочки для прямого взаимодействия с платформой и сервис REST для сетевого взаимодействия.

Ключом к удобству пользования данной платформой является предложенный разработчиками небольшой, но при этом расширяемый пакет интерактивной оболочки, помогающий пользователям управлять платформой.

4.2. Языки и инструменты

На самом деле проект AspectJ 1.8 относится как к языковым реализациям, так и к инструментальным Он представляет собой бесшовное аспектно-ориентированное расширение для языка Java. Это совместимая платформа Java для более легкого обучения и использования языка.

AspectJ 1.8.0.M1 является восьмым компилятором Java. В чем причина изменений в текущей версии? Некоторые пользователи выбирают использование AspectJ-библиотек даже несмотря на то, что в их источнике нет конструкций языка Java8. Компилятор Eclipse JTD (Eclipse 4.3) включает некоторые изменения, призванные облегчить такое использование. Эти изменения необходимы, так как классы 1.8 включают в себя мета-данные, для обработки которых компилятор 1.7 не предназначен. Например, внедренный метод реализован в интерфейсе. Для того чтобы поддержать этот способ работы, AspectJ нуждался бы в обновлении до компилятора Eclipse 4.3. Тем не менее, выполняющиеся обновления компилятора в AspectJ не тривиальны. Чтобы избежать обновления до 4.3 и дальнейшего обновления до компилятора Java8 разработчики решили сразу перейти к Java8, который уже содержит все необходимые изменения.

4.3. Моделирование

Представляющий категорию моделирования проект MoDisco является расширяемой платформой для конкретизации управляемых моделями

решений, поддерживающей случаи модернизации программного обеспечения (техническая миграция, улучшение программного обеспечения, создание документации, качество страхования и т.д.).

Унаследованные системы охватывают большое количество технологий, делающих разработку инструментов для развития устаревших проблем утомительной и трудоемкой задачей. Поскольку проекты по модернизации сталкиваются с обеими комбинациями технологий и различными ситуациями модернизации, управляемые моделями подходы и инструменты вводят уровень необходимой абстракции для создания зрелых и гибких решений модернизации.

Данная реализация идет на благо нескольким проектам Эклипс (по большей части проектам по моделированию). Существует историческая связь между проектом EMF Facet (ответвление MoDisco). Некоторые компоненты MoDisco перемещаются в EMF Facet.

Гарантия качества: всегда подтверждена, вне зависимости от того, отвечает ли существующая система необходимым требованиям (обнаружение анти-паттернов в существующем коде и вычисление метрик).

Документация: извлечение информации из существующей системы для облегчения понимания какого-то одного аспекта системы (структура, поведение, постоянство, поток информации и т.д.).

Улучшение: трансформация существующей системы для объединения с лучшими нормами и шаблонами кодировки.

Миграция: преобразование существующей системы для изменения компонента, платформы, языка или его архитектуры.

Все эти проекты носят сугубо прикладной характер, чаще всего используются в коммерческой сфере. Например, язык Java известен тем, что

широко используется в создании приложений и инструментов, которые мы используем ежедневно, и не только в рамках бизнеса.

ЗАКЛЮЧЕНИЕ

Нашей целью было рассмотреть, как возможно применение в реальной жизни логического программирования с ограничениями. Мы показали, что программирование, построенное на методе логического вывода и правила резолюции может не только нести вклад в науку искусственного интеллекта, но и использоваться в нашей повседневной жизни. Покупка проекта Эклипс компанией Cisco показала, что данная парадигма востребована не только в узких научных кругах, тем более эта компания специализируется на коммерческих и бизнес-проектах.

Рассмотренные нами в ходе исследования пункты, а именно:

- экспертные системы и их роль в нашей жизни;

- структура экспертных систем;

- элементы логики в структуре интеллектуальных систем;

- логический вывод;

- правило резолюции;

- язык логического программирования Пролог;

- программирование в ограничениях;

- логическое программирование в ограничениях;

- особенности языка Эклипс;

- реализация проектов Эклипс,

дали нам понять, что исследования, изначально родившиеся в рамках научных исследований, не обязаны таковыми оставаться. Междисциплинарность области искусственного интеллекта говорит сама за себя, и результаты этого вы видим каждый день своими глазами, более того –

активно с ними взаимодействуем, и многие реализации подобных исследований значительно облегчают нашу жизнь.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. *Гаврилова Т.А., Хорошевский В.Ф.* Базы знаний интеллектуальных систем. — СПб.: Питер, 2001.
2. *Доорс Дж., Рейблейн А.Р., Вадера С.* Пролог – язык программирования будущего. – Москва: «Финансы и статистика», 1990.
3. *Компьютеры пятого поколения* [Электронный ресурс] // Википедия: свободная энцикл. – Электрон. дан. – URL: http://ru.wikipedia.org/wiki/Компьютеры_пятого_поколения (дата обращения: 01.05.2016).
4. *Новикова В.А., Андреева Е.Ю, Туйкина Д.К.* Искусственный интеллект и экспертные системы. [Электронный ресурс]: Учебное пособие. Курск: КГУ, 2004. – 45 с. URL: http://expro.ksu.ru/materials/ii_i_es/book.html#point1.3 (дата обращения: 16.04.2016)
5. *Национальный открытый университет ИНТУИТ* [Электронный ресурс]: Шрайнер П. Основы программирования на языке Пролог [образовательный курс]: Лекция 1: Введение в язык логического программирования Пролог. URL: http://www.intuit.ru/studies/courses/44/44/lecture/1309%3Fpage%3D1 (дата обращения: 22.04.2016)
6. *Рассел С., Норвиг П.* Искусственный интеллект: современный подход / Пер. К. А. Птициной - М.: "Вильямс", 2006.
7. *Рефал* [Электронный ресурс] // Википедия: свободная энцикл. – Электрон. дан. – URL: http://ru.wikipedia.org/wiki/РЕФАЛ (дата обращения: 01.05.2016).
8. *Робинсон Дж.* Машинно-ориентированная логика, основанная на принципе резолюции / Пер. Ю. В. Матиясевича // Кибернетический сборник. 1970. №5. С. 194-218.

9. *Снобол* [Электронный ресурс] // Википедия: свободная энцикл. – Электрон. дан. – URL: http://ru.wikipedia.org/wiki/Снобол (дата обращения: 01.05.2016).
10. *Солдатова О.П., Лёзина И.П.* Логическое программирование на языке Visual Prolog: учебное пособие – Самара: СНЦ РАН, 2010.
11. *Строки* [Электронный ресурс] // Википедия: свободная энцикл. – Электрон. дан. – URL: http://ru.wikipedia.org/wiki/Строковый_тип (дата обращения: 04.05.2016).
12. *Хоггер К.* Введение в логическое программирование. – М.: Мир, 1988.
13. *Apt K., Wallace M. G.* Constraint logic programming using ECLiPSe. – Cambridge University press, 2007.
14. *Bergin T.J., Gibson R.G.* The birth of Prolog // History of Programming Languages. - ACM Press/Addison-Wesley, Reading, Massachusetts, pp. 331-367.
15. *ECLIPSE*: [Электронный ресурс]. URL: https://www.eclipse.org/ (Дата обращения: 04.05.2016).
16. *Kowalski R.* Predicate logic as programming language // Information Processing 74 – North-Holland Publishing Company, 1974.
17. *LISP* [Электронный ресурс] // Википедия: свободная энцикл. – Электрон. дан. – URL: http://ru.wikipedia.org/wiki/LISP (дата обращения: 01.05.2016).
18. *Niederlinski A.* A gentle guide to constraint logic programming via ECLiPSe. – Gliwice, 2014.
19. *Sutherland I.E.* Sketchpad, A Man-Machine Graphical Communication System // University of Cambridge as Technical Report, c.17.